

Checkers - a web app for playing checkers

Applicazioni e Servizi Web

Manuele Pasini - 0000926769 {manuele.pasini@studio.unibo.it}
Luca Salvigni - 0000952607 {luca.salvigni7@studio.unibo.it}

June 15, 2022

0.1 Introduzione

Negli ultimi due anni, supportata in particolar modo dalla diffusione della pandemia di COVID-19 e dall'uscita sulla piattaforma di streaming video Netflix della serie televisiva "La Regina degli Scacchi" il numero di giocatori di scacchi è cospicuamente aumentato, soprattutto per quanto riguarda il gioco online. Al contrario delle piattaforme web sulle quali è possibile registrarsi e giocare a scacchi, non esiste la corrispondente versione con il gioco della dama, escluse versioni in cui l'unica funzionalità possibile è quella di giocare una partita. L'idea alla base di questo progetto è quella di costruire una semplice e funzionale piattaforma web che permetta la formazione di un ecosistema di utenti che possano collettivamente giocare e rimanere in contatto attraverso il gioco della dama; si è dunque giunti alla seguente formalizzazione del progetto:

Descrizione progetto

Il progetto si intitolerà Checkers e rappresenterà una versione multiplayer online del gioco da tavolo Dama. Gli utenti avranno la possibilità di registrarsi al sito creando un proprio account. Una volta eseguito l'accesso, sarà possibile effettuare una partita contro un altro giocatore online mediante un meccanismo di ricerca o tramite invito. Una volta che due giocatori si trovano in una lobby è possibile avviare una partita e sarà messa a disposizione una chat di gioco per lo scambio di messaggi. Conclusasi la partita, verranno aggiornate le statistiche di gioco di ogni giocatore, e verranno assegnati dei punti in base alle performance; questi punti potranno essere utilizzati per stilare una classifica generale dei giocatori.

Feature principali:

- registrazione degli utenti;
- creazione di lobby di gioco a 2 giocatori;
- possibilità di invitare altri utenti alla lobby;
- visualizzazione profilo utente;
- gestione della partita;
- gestione chat di gioco;
- classifica globale (statistiche, punti, ...).

A seconda dei limiti di tempo potrebbero essere integrate le seguenti funzionalità aggiuntive:

- chat globale;
- meccanismo di gestione livello e consegna rewards.

0.2 Requisiti

Il sistema nasce sulla base del requisito principale dell'essere in grado di realizzare il più semplice e funzionale ecosistema di giocatori di dama, in particolare la piattaforma deve offrire un'esperienza che sia comprensibile e fruibile da qualunque tipologia di utente; prendendo in esame il caso del gioco degli scacchi (il quale è più diffuso ed è possibile uno studio più approfondito del contesto), è possibile notare come l'età in cui i giocatori professionisti raggiungono il grado di Gran Maestro sta progressivamente diminuendo¹ stabilizzandosi al momento attorno ai vent'anni ed allo stesso tempo è possibile individuare l'età di picco dei giocatori professionisti in un range di età che va dai trentacinque anni ai quarantacinque anni.[1] Costruita una baseline riguardante l'età target del gioco degli scacchi si è provveduto a trasporre queste informazioni e dedurre assunzioni sul gioco della dama.

Per quanto esista una scena professionale del gioco della dama questa è certamente molto inferiore a livello di estensione e diffusione rispetto a quella degli scacchi, così come inferiore è la complessità intrinseca del gioco e la durata media delle partite; infine, tenendo in considerazione che le statistiche prese in esame fanno riferimento a giocatori professionisti che non sono il target d'utenza dell'applicazione, si sono individuate di una serie di caratteristiche che potrebbero essere presenti nell'utente di Checkers:

- inesperienza nell'utilizzo di tecnologie informatiche: per la semplicità e la rapidità del gioco è possibile considerare come utente target anche persone la cui età statisticamente coincide con una scarsa conoscenza informatica;
- breve tempo a disposizione: la rapidità del gioco è un fattore influente che deve essere consolidato attraverso una rapidità nell'utilizzo dell'applicazione, misurabile nel tempo/numero di procedure necessarie per iniziare una partita dal momento in cui l'utente accede al servizio;
- internazionalità dei giocatori: vista la bassa diffusione del gioco è opportuno allargare il bacino d'utenza realizzando un applicazione che abbatta, quantomeno in parte, le barriere linguistiche; sulla base di questa considerazione si è scelto di realizzare la versione internazionale di dama, che segue le seguenti regole:
 - La scacchiera è composta da 20 caselle per lato, delle quali solamente quelle nere, che corrispondono alla metà, sono utilizzabili;
 - il sistema di coordinate utilizzato è diverso da quello degli scacchi, ogni casella è numerata progressivamente all'interno del range [1,50], dove la casella numero uno è quella collocata in alto a sinistra della scacchiera, ipotizzando pedine bianche nella parte bassa della scacchiera;

¹eta media per diventare Gran Maestro



Figure 1: Sistema di coordinate

- le pedine possono muoversi diagonalmente e la cattura di una pedina avversaria è sempre obbligatoria, così come obbligatoria è la cattura del maggior numero di pedine possibili; al contrario della dama italiana le pedine possono catturare in ogni direzione, persino muovendosi indietro, movimento possibile solamente in questa casistica.
- una pedina che raggiunge il bordo opposto della scacchiera diventa “re” perdendo il vincolo di movimento in avanti; un re può muoversi in qualunque direzione; in particolare se il movimento di un re comporta una cattura, tale movimento può avere lunghezza indefinita di x caselle.

Sulla base di questa breve profilazione sono stati identificati tre requisiti qualitativi dell'applicazione:

1. **intuitività:** l'applicazione deve colmare il gap di conoscenze tecnologiche presente in una fetta degli utenti dell'applicazione, le procedure devono essere rapidamente comprensibili ed attuabili anche da chi non ha particolari capacità in ambito informatico;
2. **semplicità:** corollario dell'intuitività, il sistema deve essere dotato di poche e semplici procedure, che permettano un utilizzo intuitivo del sistema; proprietà che fa riferimento anche all'aspetto grafico del sistema: questo deve agevolare l'esperienza di gioco e non appesantirla, deve permettere agli utenti di concentrarsi sulle partite senza distrazioni o grafiche esose.

3. **velocità**: ogni procedura attuabile all'interno dell'applicazione deve essere svolta con un numero di click non superiore a cinque.

Infine si è proceduto ad identificare i requisiti funzionali del sistema:

1. procedura di autenticazione;
2. possibilità di personalizzazione del profilo utente;
3. possibilità di creare una partita;
4. possibilità di visualizzare le partite e partecipare ad una di queste;
5. possibilità di invitare un amico;
6. possibilità di visualizzare il proprio storico delle partite;
7. presenza di un meccanismo di competizione.

I requisiti uno e sette sono stati ulteriormente esplosi e sono stati individuati i seguenti requisiti non funzionali:

1. 1.1: gli utenti devono poter essere in grado di registrarsi ed accedere al sito attraverso una procedura sicura; ulteriormente, sarà necessario un meccanismo di autenticazione tra la parte client e la parte server dell'applicazione;
2. 7.1: si è ritenuta necessaria la presenza di un meccanismo che possa portare gli utenti a non considerare l'applicazione una meta di passaggio ma che cerchi di stabilizzare la loro presenza all'interno del sistema, per questo si è deciso di introdurre un meccanismo di competizione identificato da:
 - stelle: rappresentano i punti di un utente, la vittoria di una partita corrisponde ad un guadagno di 100 stelle mentre una sconfitta comporta una perdita di 50 stelle;
 - leaderboard: la presenza di una classifica costantemente aggiornata aggiunge competizione e stimola gli utenti a migliorare il proprio risultato utilizzando l'applicazione;
 - storico partite: ogni utente dovrà essere in grado di visualizzare l'esito delle sue partite precedenti.

0.3 Design

Alla luce di quanto definito nelle fasi precedenti del progetto, sono state individuate all'interno del sistema due macro entità: **backend** e **frontend**.

Il primo comprende tutta l'architettura server-side di un'applicazione web mentre il secondo rappresenta la parte client-side. Ulteriormente il backend è stato scomposto in tre moduli realizzati sotto forma di microservizi: CommunicationService, UserService, GameService.

- gestione degli utenti e dei loro profili ⇒ **UserService**;
- gestione delle partite ⇒ **GameService**;
- gestione della comunicazione con i client ⇒ **CommunicationService**.

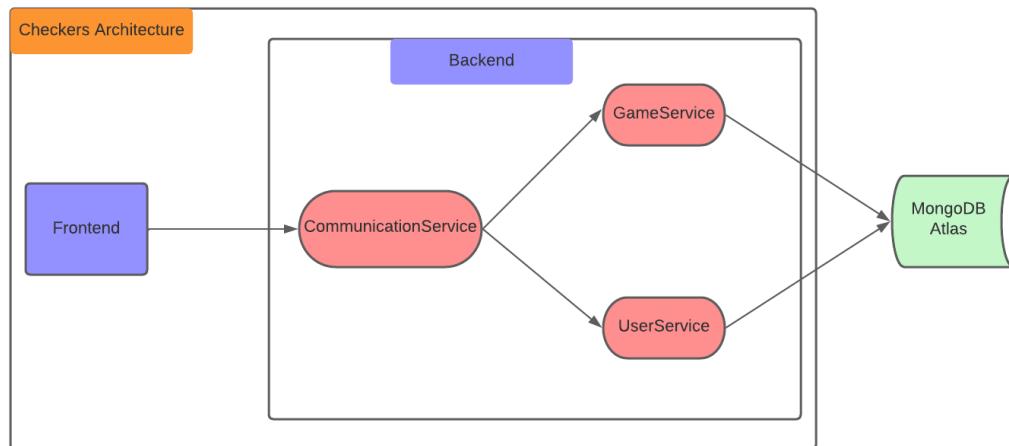


Figure 2: Esploso dell'architettura backend all'interno dell'ecosistema Checkers

0.3.1 Backend

In riferimento ai tre microservizi componenti il backend, l'idea alla base della scomposizione è derivante dalle fasi precedenti del progetto: la necessità di individuare e scomporre la parte server-side del sistema in componenti modulari ed indipendenti tra loro. Si è dunque giunti all'individuazione di tre funzionalità principali modellate tramite i conseguenti microservizi:

Infine, come evidenziato in figura 0.3, la proprietà di persistenza del sistema è garantita attraverso l'utilizzo di MongoDBAtlas, servizio che offre l'utilizzo di un database non relazionale tramite cloud; questa soluzione permette una forma

di "outsourcing" della parte di storage dell'applicazione garantendo trasparenza rispetto al funzionamento del resto del sistema e, di conseguenza, la non necessità della gestione dello storage in termini di sicurezza, disponibilità e più in generale di gestione diretta: l'accesso al DB viene gestito attraverso le apposite API mentre la gestione ed il monitoraggio possono essere fatti attraverso l'apposita interfaccia web messa a disposizione dal servizio stesso.

0.3.2 Structure

I microservizi sono stati realizzati utilizzando Node.js e Express e condividono la stessa struttura:

- controllers: contiene la logica applicativa del servizio;
- models: definisce la struttura dei dati necessari al funzionamento di tale servizio: dati strutturati per la comunicazione con il database oppure strutture dati complesse utilizzate internamente al servizio;
- routes: contiene i metodi esposti dalle API REST;
- test: contiene i test unitari del microservizio;
- index: file principale utilizzato per inizializzare il singolo microservizio.

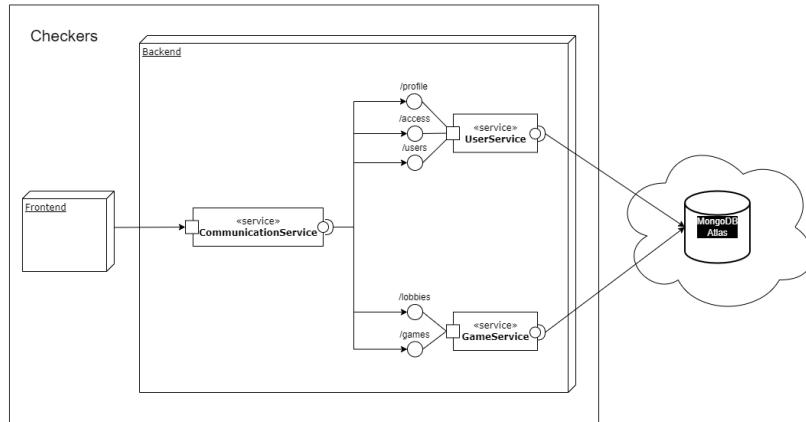


Figure 3: Diagramma dei componenti rappresentante i servizi interni al backend.

GameService e UserService sono componenti passivi e stateless del sistema; il loro comportamento è rispondere alle interrogazioni del CommunicationService ogni volta quest'ultimo necessita di interrogarli. Interagiscono unicamente con CommunicationService.

Per garantire la natura stateless di questi due servizi, si è deciso di pagare il prezzo di query frequenti sul database: prendendo il caso del GameService, ogni partita di dama è identificabile da una stringa FEN che rappresenta la posizione delle pedine sulla scacchiera: piuttosto di mantenere in locale una lista delle partite in corso, GameService associa ad ogni partita creata un identificativo e salva il FEN corrispondente a tale partita sul database; ogni qual volta uno dei due giocatori coinvolti in quella partita x compie una mossa, GameService si occupa di recuperare la partita x dal database ed aggiornare il FEN di quella determinata partita sulla base della mossa appena compiuta da uno dei due giocatori.

CommunicationService

Rappresenta il gateway del backend, la parte del sistema che si occupa di gestire le comunicazioni con i client e la logica di gestione di questi. Dal momento in cui un client si connette al sistema al momento in cui questo si disconnette è compito del CommunicationService garantire che questo possa svolgere tutte le operazioni per le quali è autenticato.

Sono compiti del CommunicationService:

- gestire connessione e disconnessione client;
- organizzare i client in lobby;
- effettuare access control;
- gestire la logica applicativa dell'applicazione.

In particolare l'ultimo punto certifica la centralità del servizio CommunicationService: è il componente core e l'unico proattivo del backend, dovendo tutte le comunicazioni passare attraverso esso, non si è ritenuto conveniente spostare la logica applicativa su un altro servizio. Questo servizio si appoggia al GameService per la gestione delle singole partite di dama e al UserService per la gestione degli utenti e delle loro proprietà, oltre che per verificare l'integrità dei client e la corrispondenza tra identità fisica (socket) ed identità digitale del client (autenticazione). E' inoltre compito di questo servizio garantire la disponibilità del sistema intercettando e gestendo eventuali errori garantendo dunque trasparenza agli utilizzatori del sistema.

Come visibile in figura 0.3.2, è la parte del sistema che comunica con il frontend e di conseguenza con i client. Svolge la funzione di gateway dell'applicazione, il "middleware" tra gli utenti che utilizzano il sistema (attraverso il Frontend) ed il Backend dell'applicazione e si occupa di soddisfare le richieste dei primi; al contrario dei restanti due microservizi, il CommunicationService è un componente attivo e stateful, mantiene una lista aggiornata degli utenti connessi e si

occupa della gestione della loro esperienza sul sistema, contattando a necessità il UserService ed il GameService.

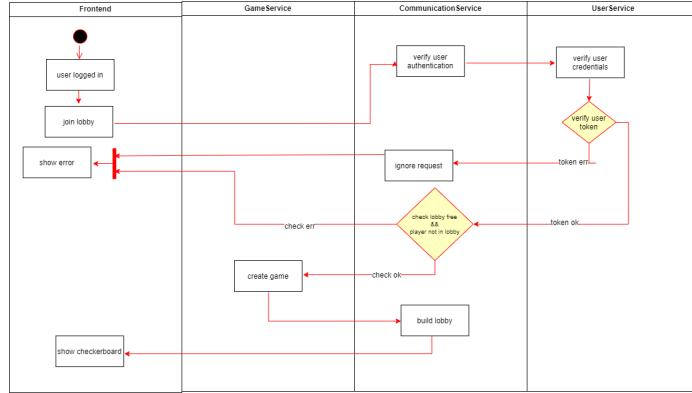


Figure 4: Diagramma di attività mostrante come i servizi dialogano tra di loro durante la procedura di login.

In figura 0.3.2 è possibile vedere, nel contesto del tentativo di un client di iniziare una partita con un secondo client, la centralità del servizio CommunicationService e la passività dei restanti due, i quali rispondono solamente alle interrogazioni del primo.

UserService

Rappresenta il servizio che si occupa della gestione degli utenti, è il servizio dedicato alla connessione alla parte del database che modella gli utenti dell'applicazione; è un servizio reattivo che risponde alle interrogazioni del SocketService. In particolare, è la parte del sistema che si occupa di garantire l'integrità dei client e verificarne l'autenticazione. Ogni qual volta un utente effettua un'azione per la quale è necessaria autenticazione, il SocketService contatta questo servizio che verifica l'autenticazione del client attraverso l'utilizzo di JSON Web Token².

E' inoltre un componente stateless, la persistenza del flusso dati che passa da questa entità è garantita da MongoDB, sul quale vengono salvate le uniche informazioni necessarie al UserService per un corretto funzionamento del sistema. Tale proprietà garantisce una maggiore scalabilità e robustezza al sistema, un'eventuale interruzione del servizio potrebbe essere gestita senza la necessità di dover recuperare il lavoro che stava svolgendo il servizio al momento dell'interruzione.

Come visibile in figura 0.3.2, il servizio espone tre endpoint:

- **/profile:** comprende tutte le richieste inerenti ad un singolo utente e la gestione del profilo di questo;

²<https://jwt.io/>

- **/users**: comprende tutte le richieste inerenti a più utenti, come ad esempio la classifica globale;
- **/access**: la parte più critica del servizio, comprende tutte le richieste inerenti il controllo degli accessi degli utenti all'interno del sistema;

GameService

Rappresenta il servizio che si occupa della gestione delle singole partite di dama, dalla loro creazione alla terminazione, e dell'interazione con la porzione di database che modella le partite tra utenti. Specularmente al UserService, anche questo servizio è di natura reattiva e risponde alle interrogazioni del SocketService. Il servizio espone le seguenti API: allo stesso modo del UserService, anche il GameService è un'entità stateless.

Come visibile in figura 0.3.2, il servizio espone due endpoint:

- **/game**: comprende tutte le richieste inerenti ad una singola partita di dama e la gestione di questa, come ad esempio l'inizializzazione di una partita e lo spostamento di una pedina da una casella x ad una casella y ;
- **/lobbies**: comprende tutte le richieste inerenti più partite, come ad esempio la richiesta di ricercare tutte le partite di un determinato utente z ;

0.3.3 Interaction

E' possibile individuare all'interno del sistema quattro entità che comunicano tra loro: i tre servizi componenti il backend e l'entità umana, che comunica con la parte di backend del sistema attraverso un client, e la parte frontend. I flussi di comunicazione sono due:

- **client - CommunicationService**: rappresenta il flusso di comunicazione che intercorre tra il client utilizzatore del sistema ed il backend del sistema, parte dal momento in cui un utente effettua un'azione sul frontend al momento in cui tale azione, tradotta in richiesta dal frontend, raggiunge il CommunicationService. Per questo flusso di comunicazione si è scelto di utilizzare la tecnologia HTTP, in particolare la scelta è ricaduta su Socket.IO³.
- **CommunicationService - UserService/GameService**: rappresenta il flusso di comunicazione che intercorre tra i servizi che compongono il backend, racchiude tutte le comunicazioni dal momento in cui una richiesta arriva al CommunicationService fino al momento in cui tale richiesta viene processata e generata una risposta da inviare al client.
Per questo flusso di comunicazione si è scelto l'utilizzo della tecnologia HTTPS, implementata tramite axios⁴.

³<https://socket.io/>

⁴<https://axios-http.com/docs/intro>

Segue l'immagine rappresentante il flusso di comunicazione tra i servizi nello stesso contesto della figura 0.3.2

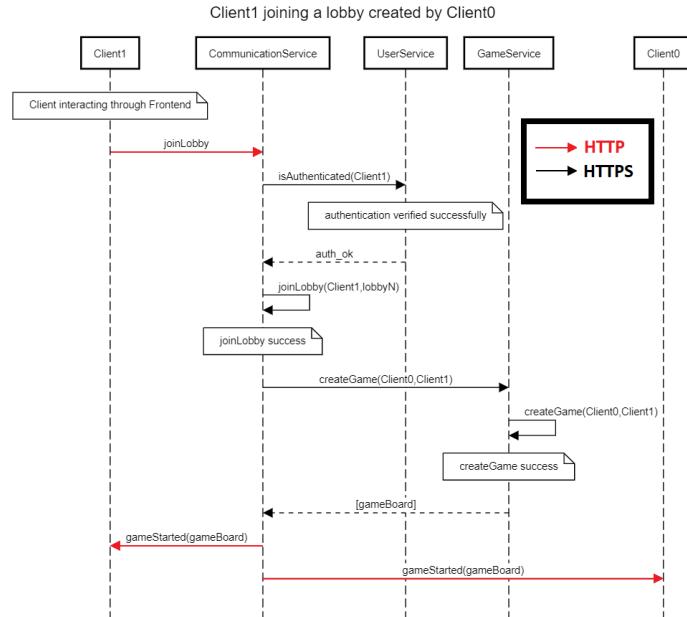


Figure 5: Caso applicativo di interazione tra client ed i tre servizi

HTTP/HTTPS

Axios⁵ è una libreria che mette a disposizione un client in grado di effettuare richieste HTTP ed HTTPS; per la seconda opzione è necessario l'utilizzo di un certificato valido; visto il contesto accademico di questa applicazione ci si è limitati a "costruire" un piccolo environment fittizio all'interno del quale una certification authority fittizia possiede un certificato auto-firmato con il quale sono stati firmati tre certificati, ognuno assegnato ad uno dei microservizi componenti il backend.

Ogni qual volta UserService e GameService ricevono una richiesta HTTPS, provvedono a verificare se il certificato del richiedente è valido, effettuando la procedura di verifica utilizzando la chiave pubblica della CA fittizia. In questo modo, UserService e GameService risponderanno solamente alle richieste di quelle entità che hanno un certificato valido, garantendo la non possibilità per una terza parte di bypassare il CommunicationService (unico altro possessore di un certificato valido) e tentare di comunicare direttamente con GameService ed UserService.

Sarebbe stato ideale estendere la comunicazione HTTPS anche tra i client del sistema ed il CommunicationService ma durante la fase di progettazione si sono

⁵<https://axios-http.com/docs/intro>

associati i seguenti problemi a tale realizzazione, che hanno portato il team a propendere per una connessione HTTP tra frontend e backend:

- necessità di installare il certificato fittizio in ogni browser che volesse lanciare l'applicazione;
- necessità di far riconoscere al browser un certificato auto-firmato come autorevole;

0.3.4 Frontend

Il frontend è la parte del sistema che gestisce l'interazione con l'utente occupandosi di ricevere dall'utilizzatore un certo flusso di informazioni. Questa sezione del software viene progettata ed implementata in modo tale da assicurare un'esperienza utente ottimale andando di fatto a rappresentare l'interfaccia utente.

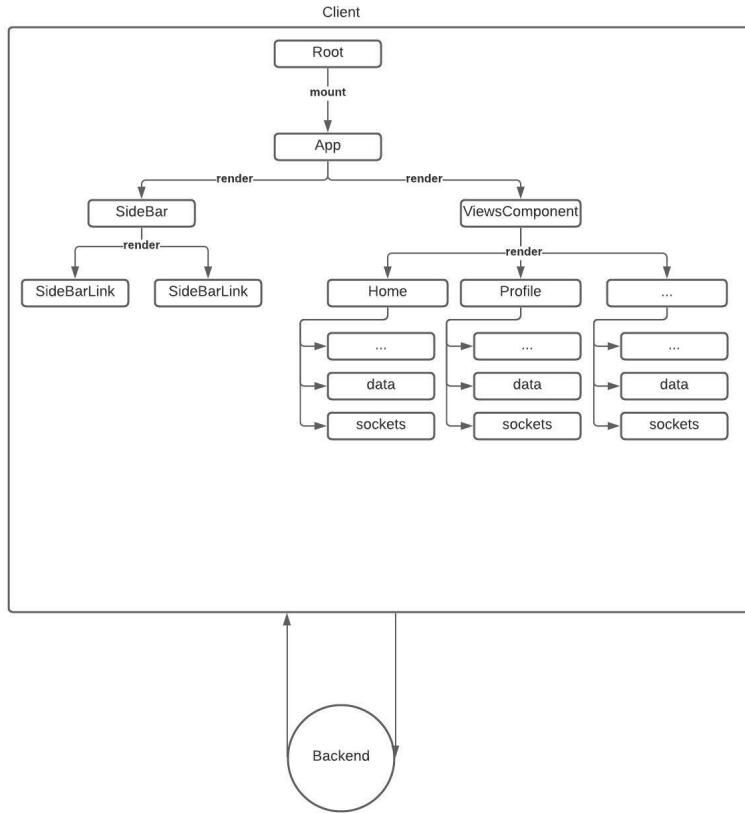


Figure 6: Rappresentazione client

Per la realizzazione della parte legata al frontend si è deciso di utilizzare **Vue**, un framework Javascript in grado di realizzare un’interfaccia utente attraverso un approccio dichiarativo. Ovvero, tramite una sua sintassi permette semplicemente di indicare cosa deve essere mostrato all’interno di una pagina Web. Sarà poi Vue stesso ad occuparsi di visualizzazione ed aggiornamento dei dati. Vue utilizza il **Virtual DOM** per capire in che modo aggiornare l’applicazione siccome l’aggiornamento continuo del DOM rappresenterebbe un’operazione molto costosa. Questo approccio permette di aggiornare prima il DOM virtuale, apportando successivamente solo le modifiche necessarie al DOM originale.

0.4 Tecnologie

Le principali scelte a livello tecnologico hanno riguardato la scelta della modalità di comunicazione tra client e server e le scelte locali di entrambe le componenti dell'applicazione; in particolare, si è optato per le seguenti tecnologie:

- **Vue 3⁶**: si è optato per utilizzare la versione più recente di Vue per quanto riguarda la costruzione della parte client del sistema: l'utilizzo di Vue è stato preferito per semplicità ad altre tecnologie come Angular e React mentre si è scelta la versione più aggiornata del framework per evitare di utilizzare materiale obsoleto;
- **Vue-router⁷**: componente ufficiale di Vue.js per il routing e la navigazione delle pagine, consentendo una mappatura nidificata delle view, una configurazione del router modulare basata sui componenti ed altro ancora;
- **Vuex⁸**: estensione per le applicazioni Vue che permette di avere quello che gli autori chiamano state management pattern, ovvero un modello per applicazioni a stato centralizzato;
- **Core-js⁹**: libreria standard modulare per JavaScript (aggiunta automaticamente dalla CLI di Vue);
- **DaisyUI¹⁰**: plugin equivalente a Bootstrap per le classi di utilità Tailwind;
- **Toasty**: plugin per la visualizzazione di Toast custom;
- **Fontawesome¹¹**: libreria per l'utilizzo di particolari tipi di icone;
- **Vitest¹²**: Vitest è un velocissimo framework di unit test basato su Vite.
- **Express 4¹³**: framework di NodeJS utilizzato per la costruzione della parte server dell'applicazione;
- **socket.io¹⁴**: per la comunicazione tra client e server si è deciso di utilizzare socket.io, libreria composta da due parti: la prima mette a disposizione una serie di funzionalità server-side, mentre la seconda mette a disposizioni funzionalità "client-side"; all'interno di questo progetto sono state direttamente utilizzate solamente le funzionalità "server-side";

⁶<https://vuejs.org/guide/introduction.html>

⁷<https://router.vuejs.org/>

⁸<https://vuex.vuejs.org/>

⁹<https://www.npmjs.com/package/core-js>

¹⁰<https://daisyui.com/>

¹¹<https://www.npmjs.com/package/@fortawesome/vue-fontawesome>

¹²<https://vitest.dev/guide/why.html>

¹³<https://expressjs.com/en/api.html>

¹⁴<https://socket.io/>

- **Vue-3-Socket.io**¹⁵: le funzionalità "client-side" del framework socket.io sono state utilizzate indirettamente attraverso la libreria Vue-Socket.io, che si occupa di facilitare l'integrazione tra socket.io e Vue 3 offrendo una modalità di gestione più user-friendly e pulita in termini di codice. Nel dettaglio ad ogni componente Vue può essere associata una proprietà "sockets" che permette all'oggetto in questione di ascoltare reagire ad una serie di eventi definiti come funzioni all'interno di questa proprietà;
- **Axios**¹⁶: per la comunicazione tra i tre microservizi che compongono il backend si è scelto di utilizzare Axios che, sfruttando il modulo nativo HTTP di NodeJS, permette ai tre microservizi di interagire tra loro attraverso richieste HTTP;
- **Crypto**: per garantire la sicurezza del processo di autenticazione degli utenti si è scelto di utilizzare la libreria nativa di NodeJS *crypto*; nel dettaglio il processo scelto per la fase di autenticazione e verifica dell'identità digitale di un utente è quello dell'utilizzo della coppia di credenziali "mail-password", all'interno del quale la password deve rispettare i seguenti vincoli:
 - numero di caratteri: [8,100];
 - numero minimo di caratteri maiuscoli: 2;
 - numero minimo di caratteri minuscoli: 2;
 - numero minimo di cifre: 2;
 - assenza di spazi bianchi;
 - numero minimo di simboli: 1.

Le password degli utenti precedentemente al salvataggio sul database vengono opportunamente hashate tramite la funzione di hash "sha-512" ed un "sale" di lunghezza 128 byte, generato randomicamente per ogni account;

- **JSON Web Tokens**¹⁷: oltre alla verifica dell'identità digitale degli utenti si è ritenuto necessario adottare un meccanismo di verifica dell'integrità dei client stessi; per questo motivo si è optato per l'implementazione di un meccanismo di autenticazione dei client verso il server attraverso l'utilizzo dei JSON Web Tokens; nel dettaglio: il servizio UserService è responsabile dell'autenticazione degli utenti e dei client, al momento dell'esecuzione del servizio questo si occupa di generare o recuperare, nel caso sia stata generata precedentemente, una stringa di lunghezza 32 bytes che permette al servizio di garantire l'autenticazione dei client; ogni qual volta un utente, attraverso l'utilizzo di un client, si connette al server ed effettua il login, lo UserService si occupa di generare un token, firmato tramite la stringa generata precedentemente, con una validità limitata nel tempo; il token

¹⁵<https://github.com/MetinSeylan/Vue-Socket.io>

¹⁶<https://axios-http.com/>

¹⁷<https://jwt.io/>

viene successivamente inviato al client che si occupa di salvarlo all'interno delle variabili di sessione e di presentarlo al server ogni qual volta voglia interagire con esso;

- **Mongoose**¹⁸: la connessione al database è gestita la libreria Mongoose, che offre una serie di API user-friendly per la gestione della connessione e delle interrogazioni al database;
- **MongoDB Atlas**¹⁹: come database si è deciso di utilizzare il servizio messo a disposizione da MongoDB Atlas, che offre la possibilità di hostare gratuitamente un database sul cloud raggiungibile attraverso una stringa di connessione ed un'autenticazione tramite credenziali. La scelta è stata fatta sulla base di tre fattori: sicurezza, persistenza, presenza di un'interfaccia grafica per la gestione e scalabilità. E' possibile monitorare lo stato del database ed il suo contenuto effettuando il login all'interno di *MongoDB Atlas* con le seguenti credenziali (rimuovendo gli apici):

```
mail = "inplrbagnngfhjphqu@nthrl.com"  
password: "123456abc*"
```

- **draughts.js**²⁰: la logica dietro alla gestione di una partita di dama è stata realizzata tramite l'utilizzo di una libreria di terze parti draught.js, per la quale ogni istanza della libreria rappresenta una partita di dama internazionale;
- **Mocha**²¹-**Chai**²²: librerie utilizzate nella fase di testing del backend.
- **Docker**: utilizzato per il deploy dell'applicazione.

¹⁸<https://mongoosejs.com/>

¹⁹<https://www.mongodb.com/atlas/database>

²⁰<https://github.com/shubhendusaurabh/draughts.js/>

²¹<https://mochajs.org/>

²²<https://www.chaijs.com/>

0.5 Codice

0.5.1 Frontend

Struttura del progetto

La struttura di file e directory è quella classica di un progetto creato tramite la Command Line Interface di Vue. Di seguito sono riportati gli elementi principali:

- **public:** cartella contenente il file HTML index.html che costituisce lo scheletro della Single Page Application, in cui poi Vue andrà ad inserire i sorgenti compilati dalla cartella src
- **src:** contiene tutti i sorgenti che la CLI di Vue compila durante la fase di build
 - **assets:** cartella contenente tutti gli assets dell'applicativo che verranno poi utilizzati dai vari component;
 - **components:** contiene tutti i single file components del progetto. I componenti al suo interno sono suddivisi a loro volta in altre directory:
 - * **boardComponents:** al suo interno vi sono i file di layout utilizzati durante una partita tra cui: **Cell**, **Chat**, **CheckerBoard**, **Player**
 - * **profileComponents:** raggruppa tutti quei componenti utilizzati per la visione della pagina legata al profilo dell'utente tra cui: **DataInfo** e **MatchInfo**
 - * **sidebarComponents:** contiene i file utilizzati per la rappresentazione della sidebar: **Sidebar** e **SidebarLink**
 - **router:** contiene un file per la gestione delle route
 - **store:** possiede un file per mantenere informazioni riguardo all'utente
 - **views:** raggruppa tutti i vari componenti che costituiscono le varie pagine dell'applicazione
 - **App.vue:** definisce la struttura della view
 - **main.js:** entry point dell'applicativo; contiene il setup e la creazione dell'istanza globale di Vue
 - **api.js:** file contenente le varie richieste che si possono effettuare al backend

Gestione delle route

Con l'utilizzo di Vue-router, grazie alla possibilità dell'utilizzo del tag **router-view** presente all'interno del componente **App**, quest'ultimo potrà inserire il contenuto della pagina corrente.

```

...
<div id="main" class="flex flex-row min-h-screen min-w-screen">
  <Sidebar @checkInvite="checkInvite" class="sidebar h-max" :invites="this.invites" />
  <div class="middle w-screen min-w-fit h-screen">
    <router-view />
  </div>
</div>

```

Le varie route sono presenti all'interno del file **index.js** della cartella **router**.

```

...
const routes = [
{
  path: '/',
  name: 'Home',
  component: Home
},
{
  path: '/profile',
  name: 'Profile',
  component: Profile
},
...
]
...

```

All'interno dei vari componenti sono presenti dei tag **router-link** che, grazie all'utilizzo dell'attributo **to**, impostano la route verso una determinata pagina.

```

<router-link to="/signup">
  <button @click="buttonClick" class="font-bold text-lg btn-link mb-3">Iscriviti</button>
</router-link>

```

Per esempio in questo caso, al click del pulsante **Iscriviti**, l'applicazione sarà reindirizzata verso la pagina di **Signup**.

State management

Alcuni dati ricevuti dal Bakend, che sono di vitale importanza per riuscire a far girare l'applicativo in maniera adeguata, sono memorizzati all'interno dello store di Vuex, la cui funzionalità è lo state management del frontend. **index.js**

```

...
export default createStore({
  state: {
    ...
  },
  getters: {

```

```

    ...
},
mutations: {
  setToken(state, token) {
    ...
  },
  unsetToken(state) {
    ...
  },
  setUser(state, user){
    ...
  },
  setInGame(state, value){
    ...
  }
}
...

```

Questo modulo è suddiviso in tre parti principali:

- state: stato memorizzato dello store che possiede dati relativi all'utente come username e mail ma anche relativi all'essere in partita o per valutare lo stato del token ottenuto dopo che la fase di login è stata effettuata con successo
- getters: proprietà per poter accedere ai dati dello store dall'esterno
- mutations: metodi utilizzati per modificare lo stato dello store

Autenticazione

La parte di autenticazione viene effettuata utilizzando come riferimento il token dell'utente presente all'interno dello store di Vuex del modulo **index.js**. Per effettuare la fase di autenticazione, l'utente dovrà inserire mail e password che verranno inviate al backend per verificare se l'autenticazione è stata effettuata correttamente. In caso negativo l'utente dovrà rieffettuare la fase di accesso, oppure si dovrà iscrivere al sito nel caso in cui non l'avesse effettuato in precedenza. In caso positivo verrà aggiornato il token presente all'interno dello store che permetterà all'utente di essere autenticato. Ogni qual volta l'utente vorrà effettuare un'operazione all'interno del sito come per esempio verificare il proprio profilo, verrà prima di tutto verificata l'esistenza del token associato a quest'ultimo e in caso negativo dovrà rieffettuare la fase di login.

Creazione di una partita

Una partita può essere svolta da solamente due utenti che sono autenticati all'interno del sito. Un giocatore può decidere di iniziare un game attraverso tre differenti modalità:

- Creando una lobby;
- Unendosi ad una lobby;
- Invitando un giocatore.

Attraverso la creazione di una lobby, l'utente dovrà prima di tutto indicare il nome della lobby e il numero di stelle massimo entro cui un giocatore vi può accedere. Confermando poi la creazione della lobby quest'ultimo verrà reindirizzato alla pagina di gioco e potrà iniziare una partita solamente quando un'altro giocatore accederà a quest'ultima.

```
...
api.build_lobby(this.$socket, lobbyName[0].value, starTextBox[0].value)
this.$router.push("/inGame")
...
```

Nel caso in cui un giocatore decida invece di unirsi ad una lobby che è già stata creata, quest'ultimo verrà prima di tutto reindirizzato ad una pagina in dove verranno mostrate tutte le lobby attualmente in attesa di un player.

```
...
api.get_lobbies(this.$socket, store.getters.user.stars)
this.$router.push("/lobbies")
...
```

Dopodichè sarà poi il player a decidere in quale lobby entrare.

```
...
api.join_lobby(this.$socket, id)
...
```

Se invece un giocatore volesse giocare con un suo amico o con un altro giocatore specifico, basterà inviare una richiesta all'opponent inserendo la sua mail

```
...
api.invite_opponent(this.$socket, opponent[0].value)
...
```

L'invito a sua volta potrà essere accettato istanziando di conseguenza la lobby nel caso in cui non ci fossero problemi o potrà anche essere rifiutato inviando la risposta di rifiuto al proprietario dell'invito stesso.

```
...
api.accept_invite(this.$socket, this.opponent_mail)
...
...
api.decline_invite(this.$socket, this.opponent_mail)
...
```

Gestione partita

Quando due giocatori sono entrambi all'interno della stessa lobby, avrà inizio la partita tramite un messaggio **gameStarted** dove al suo interno sono presenti quattro parametri fondamentali:

- lobbyId: ovvero l'identificatore della lobby stessa;
- player1: il primo giocatore che dovrà effettuare la prima mossa;
- player2: il secondo giocatore;
- board: la disposizione dei vari pezzi all'interno della board di gioco e delle possibili mosse che un pezzo potrà effettuare.

Verrà settata una variabile **playerTurn** per riuscire a gestire correttamente il turno di un giocatore, una variabile **moves** per associare i vari pezzi alle celle e una variabile **possibleMoves** che indica le mosse possibili di un giocatore specifico.

```
game_started(res) {
    // Setting all local variables for the game
}
```

Quando un giocatore dovrà effettuare una mossa, verranno prima di tutto colorate tutte le celle che potranno essere spostate all'interno della board e verso quale direzione. Nel caso in cui un giocatore possa mangiare un'altra pedina, allora verranno evidenziate solamente le celle specifiche

```
checkPossibleEat() {
    var possibleEat = []
    for(const [key, value] of Object.entries(this.myMoves)) {
        ...
    }
    return possibleEat
},
colorCells() {
    for(const [key] of Object.entries(this.myMoves)) {
        ...
    }
    if(this.playerTurn === user.mail) {
        var possibleEat = this.checkPossibleEat()
        if(possibleEat.length > 0) {
            for(let i = 0; i < possibleEat.length; i++) {
                ...
            }
            return
        }
        ...
    }
}
```

Alla selezione di un pezzo specifico questo potrà essere mosso verso una direzione specifica scelta dal giocatore tra le varie mosse possibili che gli sono concesse ed invierà al backend lo spostamento di un determinato pezzo verso una determinata cella

```
...
api.move_piece(this.$socket, this.lobbyId, this.clickedCell, cell)
...
```

Dopo che un giocatore avrà effettuato una mossa, verranno intercettati due messaggi. Il primo riguarda l'aggiornamento della board per in modo da far visualizzare all'avversario lo spostamento di una certa pedina e aggiornando di conseguenza le mosse possibili che potrà effettuare

```
update_board(res) {
    ...
}
```

Il secondo messaggio che viene inviato ai giocatori è il cambio che permetterà di gestire in modo corretto il turno, specificando chi sarà il prossimo giocatore a poter svolgere una mossa

```
turn_change(res) {
    this.playerTurn = res.next_player
    ...
}
```

Al termine di una partita, quest'ultima potrà avere quattro esiti differenti: vittoria, sconfitta, pareggio, vittoria a tavolino. Alla ricezione del messaggio **gameEnded**, verrà mostrato tramite un modale il risultato della partita per il giocatore specifico che potrà essere appunto di vittoria, sconfitta o pareggio

```
game_ended(msg) {
    // Setting game end
}
```

Un altro caso possibile è la vittoria a tavolino dove, un giocatore mentre è in partita decide per qualche motivo di uscire dalla lobby, comunicandolo al backend che poi sarà in grado di comunicare all'opponente di ciò che è avvenuto.

```
...
api.leave_game(this.$socket, this.lobbyId)
...
```

Router guards

Grazie all'utilizzo di una router guard, un giocatore potrà essere reindirizzato ad una certa pagina quando si trova all'interno di una lobby. Questa si ritiene necessaria per comunicare al backend la cancellazione della lobby quando un

utente è l'unico ad essere all'interno di quest'ultima o abbandonare la lobby se invece è in partita con un altro giocatore come citato in precedenza causando una vittoria a tavolino.

```
...
api.leave_game(this.$socket, this.lobbyId)
...

...
api.delete_lobby(this.$socket, this.lobbyId)
...

beforeRouteLeave(to, from, next) {
  if(changeLocation) {
    store.commit('setInGame', false)
    changeLocation = false
    next()
  } else {
    ...
  }
}
```

Notifiche

Le notifiche vengono gestite tramite la libreria Vue3-Socket.IO, che tramite una connessione diretta al backend, permette al client di ricevere tutte le informazioni per quanto riguarda l'esecuzione dell'applicazione. Queste notifiche vengono intercettate grazie alla proprietà **sockets** che necessita di avere al proprio interno delle funzioni i cui nomi siano identici ai messaggi inviati dal backend per essere riconosciuti. Dopo di che vengono mostrate a schermo tramite un modale, dando la possibilità all'utente di capire cosa sta succedendo.

```
sockets: {
  token_ok(res){
    ...
  },
  token_error(res) {
    ...
  },
  permit_error(error) {
    ...
  },
  server_error(error) {
    ...
  },
  client_error(error) {
    ...
  }
}
```

```

},
lobby_deleted(msg) {
    ...
},
lobby_invitation(msg) {
    ...
},
invite_accepted() {
    ...
},
invitation_declined(msg) {
    ...
},
invitation_expired(msg) {
    ...
},
invitation_timeout(res) {
    ...
},
invite_error(msg) {
    ...
}

```

Per quanto riguarda errori dovuti ad autenticazione, permessi e connessione questi vengono mostrati a schermo alla ricezione dei relativi messaggi di **token_error**, **permit_error** e **client_error**

Per la gestione degli inviti, prima di tutto un giocatore che effettua un invito, nel caso in cui non vada a buon fine gli verrà mostrato un messaggio di errore grazie alla funzione **invite_error**, altrimenti se ha successo, il destinario riceve un messaggio di **lobby_invitation** che permetterà di avvisarlo tramite un suono. Cliccando sul rispettivo invito, potrà poi scegliere se accettare o meno quest'ultimo informando il mittente tramite un messaggio di **invite_accepted** o di **invite_declined** Inoltre, un invito se rimane aperto per troppo tempo scadrà e il server invierà il rispettivo messaggio di **invitation_timeout**. Un'altra possibile situazione legata agli inviti può essere quella che un giocatore possa invitare più giocatori ed il primo che accetterà potrà entrare in partita, mentre tutti gli altri giocatori che accetteranno l'invito in seguito, riceveranno un messaggio di **invitation_expired**, informandogli che purtroppo l'invito non è più valido.

Un'altra notifica importante consiste nell'informare un utente in partita nel caso in cui il suo avversario decida di uscire tramite un messaggio di **opponent_left**, assegnandogli una vittoria a tavolino e reindirizzandolo alla pagina di **Home**

0.5.2 Backend

Gestione utenti

Gli utenti vengono memorizzati all'interno del server attraverso una mappa che mette in correlazione il client che ha aperto la connessione col server e l'utente registrato a lui associato, nella forma

```
const online_users = new BiMap() // { client_id <-> user_mail }
```

L'utilizzo di una BiMap si è reso necessario dal momento in cui non sempre il problema da risolvere è "dato il client che ha appena inviato una richiesta trovare l'utente corrispondente"; prendendo in esame la casistica in cui un utente X invita un utente Y , X conosce la mail di Y ed il server deve essere in grado di risalire dalla mail di Y al corrispondente client al quale comunicare l'invito. L'utilizzo di questa BiMap non rompe il vincolo di integrità della mappa in quanto così come l'id del socket associato ad un utente, anche le mail sono univoche all'interno del sistema. Questa scelta comporta un overhead di gestione dovuto alla BiMap che può essere vista come un wrapper su una coppia di mappe.

Gestione inviti

La gestione degli invitati a un giocatore ad un altro ha richiesto una modellazione particolare visto il numero di possibili casistiche in cui si può trovare il sistema; nel dettaglio si è deciso di permettere ad ogni utente X invitare più utenti contemporaneamente; il principio di accettazione dell'invito è FIFO, il primo tra gli utenti invitati che accetta l'invito sarà l'avversario di X ; per tutti gli altri, nel caso in cui accettino l'invito, sarà necessario mostrare un errore. Ogni invito ha una durata temporale limitata terminata la quale l'invito deve essere considerato scaduto.

```
const invitation_timeouts = new Map() // {host_id -> {opponent_id -> timeout}}
```

In questo modo, ad ogni utente X è in grado di inviare un invito ad ogni giocatore Y ed ad ogni coppia X_i, Y_i è associato un timeout che specifica la durata dell'invito. Nel momento in cui un giocatore Y_i accetta l'invito di un giocatore X_i , la mappa soprastante viene passata e vengono resettati e successivamente rimossi tutti i timeout ed i conseguenti inviti che coinvolgono X_i ed Y_i

```
// If the player invited has invited someone else, delete those invites
if(invitation_timeouts.has(Y)){
    for(const [_,invite] of invitation_timeouts.get(Y)){
        clearTimeout(invite)
    }
}
// Delete every invite made by the inviting player X
for(const [_,invite] of invitation_timeouts.get(X)){
    clearTimeout(invite)
}
```

I giocatori che hanno ricevuto un invito da X non vengono notificati della scadenza dell'invito ricevuto, tuttavia, nel caso in cui accettino un invito scaduto il server provvede a notificare il corrispondente "invitation_expired"

Gestione lobby

Le lobby sono un concetto importante all'interno del sistema, un utente può creare una lobby attendendo che un giocatore esterno entri per poi iniziare una partita oppure può invitare, tramite indirizzo mail, un utente all'interno della propria lobby. Tale concetto è stato modellato mediante l'utilizzo della funzionalità "rooms" offerta dalla libreria socket.io: è possibile organizzare le connessioni in "stanze" logiche, che facilitano la comunicazione di messaggi ai socket presenti nella stanza:

```
//client joining a lobby
client.join(lobby_id)
//sending communications to lobby
io.to(lobby_id).emit(someLobbyMessage)
```

La lobby logica è stata affiancata da un modello "fisico" di lobby, definito all'interno del CommunicationService:

```
class Lobby{

    constructor(stars,room_name,turn) {
        this.stars = stars;
        this.room_name = room_name
        this.turn = turn;
        this.tie_requests = []
        this.players = []
        this.players.push(turn)
    }
}
```

Tale modello permette al CommunicationService di avere una rappresentazione fisica di una lobby attraverso la quale facilitarne la gestione.

Gestione disconnessione

La gestione della disconnessione di un utente necessita di particolare attenzione visto il numero di situazioni diverse all'interno del quale il client si può trovare, nel dettaglio:

```
async function handle_disconnection(player){
    //Player isn't in a lobby so just disconnect it
    if(!isInLobby(player)){
        //Disconnect user
    }else{
```

```

//player is in a lobby, check if it's empty or he is in a game
if(lobby.isFree()){
    // Lobby is free, Just delete the lobby and make client leave
else{
    //Lobby is full and a game is on, need to check if it's the host or not
    if(lobby.getPlayers(0) == player){
        //Player disconnecting is the host
    else{
        //Player disconnecting is the guest
    }
    //Clear lobby room and assign points to winner and loser
}
}
}

```

Gestione asincronia

Per la gestione dell'asincronia del codice si è deciso di utilizzare il costrutto async/await offerta da Javascript: in questo modo è possibile avere codice più pulito rispetto all'utilizzo di catene di promise e callback.

```

const {data: move_result} = await
    ↳ axios.put(game_service+"/game/movePiece", {game_id:
    ↳ lobby_id, from:from,to:to})

io.to(lobby_id).emit("update_board",move_result.board)

```

0.6 Test

0.6.1 Test UserService e GameService

I servizi UserService e GameService sono stati testati tramite l'utilizzo delle librerie Mocha, Chai e Axios; in particolare per ogni servizio sono stati definiti una serie di test in grado di testare il corretto funzionamento delle funzionalità esposte da entrambi i servizi. I test si trovano all'interno delle directory di progetto dei rispettivi servizi.

File	%Stmts	%Branch	%Funcs	%Lines
All files	94.98	85.48	100	94.94
Checkers-GameService	100	100	100	100
index.js	100	100	100	100
Checkers-GameService/controllers	93.42	93.75	100	93.33
gameController.js	93.42	93.75	100	93.33
Checkers-GameService/models	100	100	100	100
gameModel.js	100	100	100	100
Checkers-GameService/routes	90.9	75	100	90.9
routes.js	90.9	75	100	90.9
Checkers-GameService/test	100	50	100	100
test.js	100	50	100	100
Checkers-GameService/test/utils	82.6	50	100	82.6
axiosRequests.js	82.6	50	100	82.6

Figure 7: Coverage di GameService

File	%Stmts	%Branch	%Funcs	%Lines
All files	95.33	88.63	100	95.31
Checkers-UserService	100	100	100	100
index.js	100	100	100	100
Checkers-UserService/controller	94.26	88.37	100	94.26
userController.js	94.26	88.37	100	94.26
Checkers-UserService/models	100	100	100	100
userModel.js	100	100	100	100
Checkers-UserService/routes	100	100	100	100
routes.js	100	100	100	100

Figure 8: Coverage di UserService

0.6.2 Test CommunicationService

CommunicationService è il servizio core dell'applicazione che si occupa di gestire i client e comunicare con i restanti due servizi; i test per questo componente possono essere definiti test d'integrazione in quanto verificano e certificano la corretta integrazione delle tre parti in gioco: client, user service e game service. I test d'integrazione sono stati effettuati attraverso l'utilizzo di **Postman**²³ e

²³<https://www.postman.com/>

delle libreria Mocha, Chai e Axios. Postman da Luglio 2021 questo tool permette di creare delle **WebSocket Request** che possano supportare la libreria **SocketIO**. Nel nostro caso ogni richiesta rappresenta un effettivo client, al cui interno vengono definiti tutti gli eventi che si possano verificare nello scambio di informazioni con il CommunicationService, come per esempio un errore nell'autenticazione, un errore nel muovere un pezzo, la conferma di autenticazione e così via. L'utilizzo di Postman permette di avere una visione d'insieme del funzionamento della componente backend del sistema e mette inoltre in evidenza eventuali problematiche logiche dell'applicazione. Per effettuare questi test, oltre alla necessità di avere Postman, è necessario che i tre servizi CommunicationService, UserService e GameService attivi e funzionanti.

File	%Stmts	%Branch	%Funcs	%Lines
All files	95.34	83.8	94.33	95.56
Checkers-CommunicationService	100	100	100	100
index.js	100	100	100	100
Checkers-CommunicationService/controller	94.9	83.45	93.18	95.14
communicationService.js	95.1	83.33	92.5	95.37
network_module.js	92.85	84.61	100	92.85
Checkers-CommunicationService/models	100	100	100	100
lobby.js	100	100	100	100

Figure 9: Coverage di CommunicationService

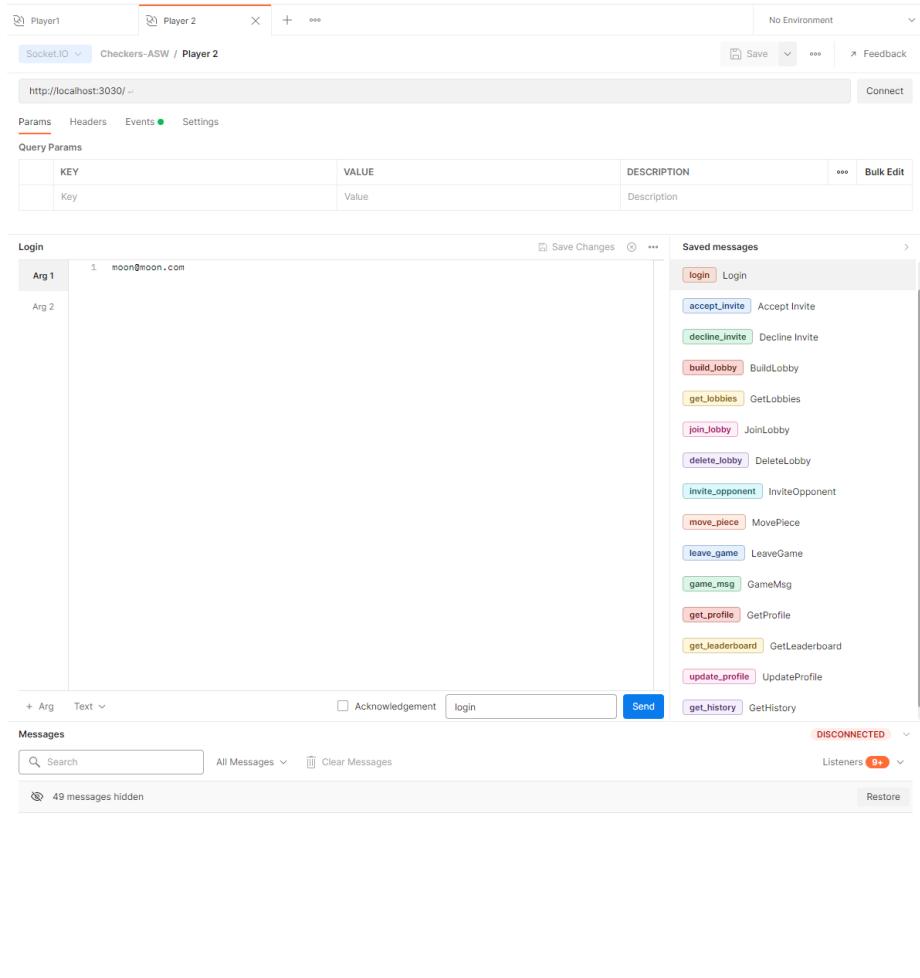


Figure 10: Messaggi che può inviare un client

The screenshot shows a configuration interface for a Socket.IO application named "Checkers-ASW / Player 2". The interface includes tabs for "Params", "Headers", "Events", and "Settings". The "Events" tab is selected, displaying a list of 30 events with their descriptions and connection settings.

Event	Description	Listen on connect
login_ok	Description	<input checked="" type="checkbox"/>
login_error	Description	<input checked="" type="checkbox"/>
login	Description	<input checked="" type="checkbox"/>
signup	Description	<input checked="" type="checkbox"/>
signup_success	Description	<input checked="" type="checkbox"/>
signup_error	Description	<input checked="" type="checkbox"/>
build_lobby	Description	<input checked="" type="checkbox"/>
lobbies	Description	<input checked="" type="checkbox"/>
permit_error	Description	<input checked="" type="checkbox"/>
token_error	Description	<input checked="" type="checkbox"/>
get_lobbies	Description	<input checked="" type="checkbox"/>
game_started	Description	<input checked="" type="checkbox"/>
server_error	Description	<input checked="" type="checkbox"/>
delete_lobby	Description	<input checked="" type="checkbox"/>
lobby_deleted	Description	<input checked="" type="checkbox"/>
invite_opponent	Description	<input checked="" type="checkbox"/>
invite_error	Description	<input checked="" type="checkbox"/>
invitation_timeout	Description	<input checked="" type="checkbox"/>
invite_accepted	Description	<input checked="" type="checkbox"/>
lobby_invitation	Description	<input checked="" type="checkbox"/>
invitation_expired	Description	<input checked="" type="checkbox"/>
decline_invite	Description	<input checked="" type="checkbox"/>
move_piece	Description	<input checked="" type="checkbox"/>
update_board	Description	<input checked="" type="checkbox"/>
game_ended	Description	<input checked="" type="checkbox"/>
client_error	Description	<input checked="" type="checkbox"/>
leave_game	Description	<input checked="" type="checkbox"/>
left_game	Description	<input checked="" type="checkbox"/>

Figure 11: Eventi che si possono scatenare durante la connessione

	Jump to latest	
① Listening to invite_accepted	11:41:06	
① Listening to invitation_timeout	11:41:06	
① Listening to invite_error	11:41:06	
① Listening to invite_opponent	11:41:06	
① Listening to lobby_deleted	11:41:06	
① Listening to delete_lobby	11:41:06	
① Listening to server_error	11:41:06	
① Listening to game_started	11:41:06	
① Listening to join_lobby	11:41:06	
① Listening to get_lobbies	11:41:06	
① Listening to token_error	11:41:06	
① Listening to permit_error	11:41:06	
① Listening to lobbies	11:41:06	
① Listening to build_lobby	11:41:06	
① Listening to signup_error	11:41:06	
① Listening to signup_success	11:41:06	
① Listening to signup	11:41:06	
① Listening to login_error	11:41:06	
① Listening to login_ok	11:41:06	
① Listening to login	11:41:06	
Connected to http://localhost:3030/	11:41:06	▼

Figure 12: Connessione di un client con il SokcetService

Login ed inizializzazione partita

Le due immagini seguenti mostrano un caso d’uso dell’interazione tra due utenti che effettuano il login e iniziano una partita sull’applicazione.

Figure 13: Login e creazione lobby e movimento di una pedina primo player

Figure 14: Login, join lobby e movimento di una pedina del secondo player

0.6.3 Test Frontend

Per quanto riguarda il Frontend sono stati effettuati dei test che permettessero di verificare il corretto rendering di tutti i componenti del sito. Ciò ha permesso quindi di verificare che le varie pagine rispondessero in modo adeguato a determinati eventi e soprattutto la reattività in base a determinate informazioni ricevute. Per la fase di testing lato Frontend sono stati utilizzati il framework Vitest ed il plugin messo a disposizione da Vue Test Utils.

File	%Stmts	%Branch	%Funcs	%Lines
All files	92.86	85.85	85.96	92.86
Checkers-Frontend	89.33	100	77.77	89.33
api.js	89.33	100	77.77	89.33
SideBar.vue	100	100	66.66	100
Checkers-Frontend/src/store	71.87	100	62.5	71.87
index.js	71.87	100	62.5	71.87
Checkers-Frontend/src/views	98.06	92.85	85.71	98.06
ErrorView.vue	100	100	100	100
Game.vue	87.57	77.77	90.9	87.57
Home.vue	99.44	95.65	80	99.44
LeaderBoard.vue	96.55	80	77.77	96.55
Lobbies.vue	100	100	100	100
LogIn.vue	100	100	87.5	100
Profile.vue	100	100	100	100
SignUp.vue	100	100	71.42	100

Figure 15: Coverage del Frontend

0.6.4 Test in fase di progettazione

Inizialmente prima di passare alla fase effettiva di sviluppo, sono stati raccolti alcuni dati fondamentali con una quantità limitata di utenti. Questa fase è molto importante per capire bene cosa fare e per cercare di minimizzare il più possibile eventuali problemi, riuscendo a capire lo scopo del progetto, individuare il target utenti che andranno ad usufruire del sito in modo da raccogliere i requisiti necessari per comprendere nel dettaglio cosa deve essere fatto. Per aiutare ancora di più questa fase, è stata svolta una parte dedicata al design grafico, dove grazie a dei mockup si è riusciti a fornire una bozza del sito, permettendo una maggior comprensione sul fatto che il sito potesse risultare soddisfacente per gli utenti e aiutando poi i programmatori nella fase di sviluppo su come dovranno essere disposti i vari contenuti all'interno dell'applicazione.

0.6.5 Test User Experience

La user experience o UX è tutto ciò che ha a che fare con l'esperienza che un utente ha vissuto nel momento in cui sta usufruendo di un determinato prodotto. Per valutare questo campo sono stati coinvolti una quantità limitata di utenti per esaminare alcune discipline fondamentali al termine della fase di sviluppo:

- **Usabilità:** il sito è risultato fin da subito semplice da navigare in quanto le pagine da visualizzare non semplici ed intuitive. Ci sono state alcune discussioni per quanto riguarda l'avvio di una partita in quanto inizialmente il pulsante di gioco era presente nella Sidebar del sito e, siccome non ha avuto molti riscontri positivi, si è optato per aggiungere questa funzionalità all'interno della pagina di Home dando una valutazione positiva dalla maggior parte degli utenti.
- **Utilità:** l'applicazione è risultata fin da subito valida per chiunque abbia voglia di fare una partita a dama. La dimensione della board è leggermente

diversa in quanto più grande, ma le regole sono sempre le stesse della dama classica.

- Funzionalità: per quanto riguarda opzioni abbastanza generali come registrazione, autenticazione, modifica dei dati profilo e altro ancora, il tutto si è dimostrato completamente funzionante senza problemi. Durante lo svolgimento di una partita ci sono stati però alcuni consigli da parte degli utenti, in quanto inizialmente non risultava molto intuitivo capire quando iniziava il turno di un giocatore. Per questo si è deciso di colorare solamente le celle delle pedine che si possono muovere al giocatore a cui toccherebbe effettuare una mossa.
- Grafica: il layout del sito è risultato piacevole agli utenti e l'utilizzo di colori scuri ha dato un riscontro positivo. Inoltre anche la scelta di usare delle piccole icone per descrivere delle azioni ha avuto riscontri molto positivi.
- Accessibilità: nell'analizzare questa disciplina, gli utenti ci hanno fatto notare che tradurre l'applicazione in inglese sarebbe stata l'opzione migliore in quanto si rende il sito accessibile ad un maggior numero di persone applicando di conseguenza questa modifica.
- Visibilità su diversi dispositivi: grazie alla possibilità di reclutare utenti diversi che potessero utilizzare diversi device, sono stati raccolti molti feedback per la visualizzazione di tutti gli elementi del sito attraverso risoluzioni di schermi differenti, adattando di conseguenza i vari problemi riscontrati dagli user.

0.7 Deployment

Il deployment dell'applicazione è stato realizzato mediante Docker: ad ognuno dei tre servizi che compongono il backend è stato associato un Dockerfile che permetta la corretta inizializzazione del servizio a cui fa riferimento; inoltre la stessa configurazione è stata realizzata per il frontend.

Per lanciare l'applicazione è sufficiente:

- collocarsi all'interno della cartella src
- copiare il file .env con tutti i secret
- eseguire il comando

```
docker-compose up
```

Per cercare di simulare il più possibile il contesto d'esecuzione dell'applicazione, i tre servizi vengono eseguiti da Docker in tre container separati ed indipendenti tra loro che comunicano grazie ad una rete interna costruita all'occorrenza da Docker:

```
services:  
  gameservice:  
    container_name: gameservice  
    image: registry.digitalocean.com/checkers-cr/checkers:gameservice  
    build:  
      context: ./Backend/Checkers-GameService  
    args:  
      - ARG_PORT=${GameService_PORT}  
      - ARG_DB_PSW=${DB_PSW}  
      - ARG_GAME_KEY=${GAME_KEY}  
      - ARG_GAME_CERT=${GAME_CERT}  
      - ARG_CERTIFICATE=${CERTIFICATE}  
    restart: always  
    ports:  
      - ${GameService_PORT}:${GameService_PORT}  
  
  userservice:  
    container_name: userservice  
    image: registry.digitalocean.com/checkers-cr/checkers:userservice  
    build:  
      context: ./Backend/Checkers-UserService  
    args:  
      - ARG_PORT=${UserService_PORT}  
      - ARG_DB_PSW=${DB_PSW}  
      - ARG_USER_KEY=${USER_KEY}  
      - ARG_USER_CERT=${USER_CERT}  
      - ARG_CERTIFICATE=${CERTIFICATE}
```

```

restart: always
ports:
- ${UserService_PORT}:${UserService_PORT}

communicationservice:
  container_name: communicationservice
  image: registry.digitalocean.com/checkers-cr/checkers:communicationservice
  build:
    context: ./Backend/Checkers-CommunicationService
    args:
      - ARG_PORT=${CommunicationService_PORT}
      - ARG_USERSERVICE_PORT=${UserService_PORT}
      - ARG_GAMESERVICE_PORT=${GameService_PORT}
      - ARG_DB_PSW=${DB_PSW}
      - ARG_COMM_KEY=${COMM_KEY}
      - ARG_COMM_CERT=${COMM_CERT}
      - ARG_GAME_KEY=${GAME_KEY}
      - ARG_GAME_CERT=${GAME_CERT}
      - ARG_USER_KEY=${USER_KEY}
      - ARG_USER_CERT=${USER_CERT}
      - ARG_CERTIFICATE=${CERTIFICATE}
      - ARG_USER_SERVICE=${USER_SERVICE}
      - ARG_GAME_SERVICE=${GAME_SERVICE}
  restart: always
  ports:
- ${CommunicationService_PORT}:${CommunicationService_PORT}

frontend:
  container_name: frontend
  image: registry.digitalocean.com/checkers-cr/checkers:frontend
  build:
    context: ./Frontend/Checkers-Frontend
    args:
      - ARG_VUE_APP_COMMUNICATION_SERVICE=${VUE_APP_COMMUNICATION_SERVICE}
  restart: always
  ports:
- 8000:80

```

Se non si vuole usufruire del deployment, è possibile eseguire l'applicativo nella maniera primordiale lanciando singolarmente i tre servizi del Backend:

- UserService:

- Spostarsi nella cartella Checkers-UserService
- Copiare il file .env
- Eseguire la coppia di comandi nel seguente ordine

```
npm install  
node index.js
```

+

- GameService:

- Spostarsi nella cartella Checkers-GameService
- Copiare il file .env
- Eseguire la coppia di comandi nel seguente ordine

```
npm install  
node index.js
```

- CommunicationService:

- Spostarsi nella cartella Checkers-CommunicationService
- Copiare il file .env
- Eseguire la coppia di comandi nel seguente ordine

```
npm install  
node index.js
```

- Frontend:

- Spostarsi nella cartella Checkers-Frontend
- Eseguire i comandi nel seguente ordine

```
npm install  
npm run lint  
npm run serve -- --port X
```

dove X è la porta in cui si vuole far girare il client.

0.8 Conclusioni

La realizzazione di un'applicazione web tramite l'utilizzo dello stack MEVN è risultato molto più intuitivo rispetto all'utilizzo di un approccio tramite XAMPP. La principale motivazione di ciò riguarda soprattutto il poter utilizzare il linguaggio Javascript sia per lo sviluppo del frontend sia per lo sviluppo del backend grazie a Node.js. Un grande pro dell'utilizzo di Node.js e del suo packet manager è la grandezza della community dietro che mette a disposizione un numero illimitato di librerie per quasi ogni esigenza, facilitando lo sviluppo e permettendo di raggiungere standard di qualità che richiederebbero molto più lavoro. Unico contro la curva di apprendimento del linguaggio e della metodologia di sviluppo che ha richiesto inizialmente una discreta quantità di tempo, recuperato successivamente tramite la fluidità dell'approccio. Infine ci sono state alcune difficoltà in fase di sviluppo dovute al fatto della scelta di utilizzare Vue 3 in quanto la documentazione ed in generale il supporto all'interno della community è risultato più limitato rispetto a Vue 2. Ci riteniamo comunque soddisfatti del risultato ottenuto.

0.8.1 Sviluppi futuri

Il lavoro svolto finora ci ha permesso di attuare una trasposizione di Checkers che, dal punto di vista delle funzionalità, è identico alla versione da tavolo della dama internazionale. Pertanto, in una prospettiva futura, gran parte degli aspetti su cui si basa il progetto possono evolvere come per esempio: aggiornare l'infrastruttura, fare affidamento sul livello di personalizzazione data dall'utente, mettere a disposizione nuove funzionalità. Sarebbe inoltre importante in una versione futura andare ad implementare ulteriori meccanismi di fault-tolerance che possano rendere il più trasparenti possibili eventuali interruzioni del servizio ai clienti. Alcuni tra i possibili lavori futuri sono:

- Aggiungere la possibilità ad un giocatore disconnesso per vari motivi di poter riconnettersi ad una lobby entro un tempo limite.
- aggiungere la possibilità di recuperare le partite in corso perse durante un'interruzione del servizio (il sistema si presta già a questa funzionalità visto il salvataggio di ogni partita sul DB etichettate come "terminata" ed "in corso").
- aggiungere un meccanismo di fault-tolerance che permetta al sistema di intercettare un eventuale interruzione di uno dei servizi e mettere in stallo l'operatività del sistema (es: un meccanismo simile a quello degli heartbeat utilizzato da Hadoop)
- Garantire più libertà di personalizzazione all'utente, per esempio cambiare colore delle pedine, cambiare colore della board di gioco o cambiare il tema dell'applicazione. Modifiche che non impattano le dinamiche di gioco ma che possono garantire una user experience migliore.

- Pubblicazione del gioco online;
- Aggiunta di nuove modalità particolari di gioco legate alla Dama come per esempio Trapdoor Checkers

Screenshot applicazione

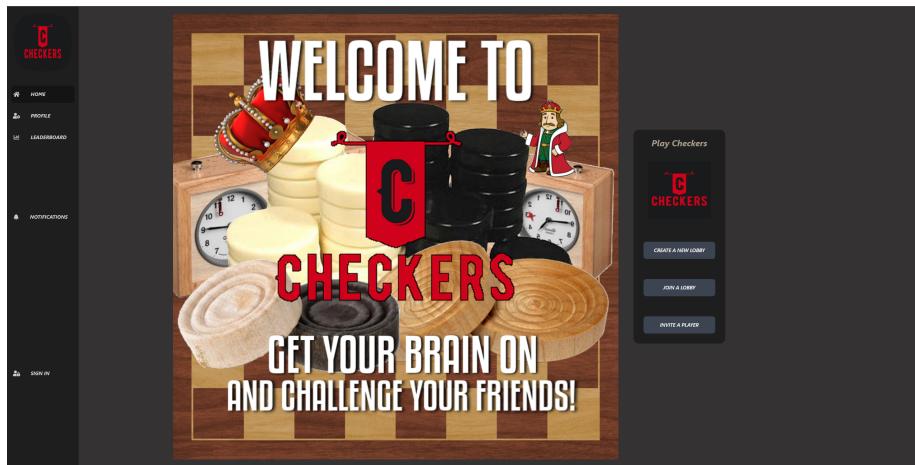


Figure 16: Pagina di Home

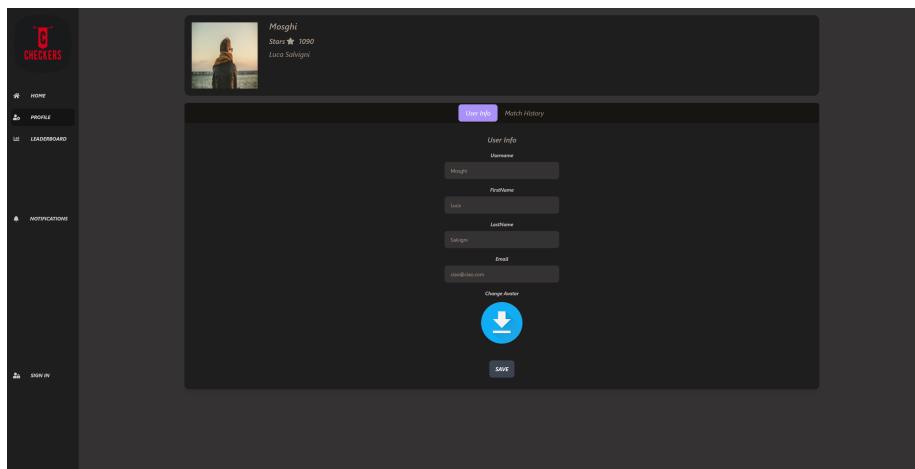


Figure 17: Pagina del profilo utente

Global leaderboard					
POSITION	NAME	STARS	GAMES	WINS	DETAILS
01	Berlin123	4000	50	52	?
02	Mann	1000	50	47	?
03	Negishi	1000	50	41	?
04	Test2	200	5	0	?
05	AG2	100	42	5	?
06	Davidef	100	3	2	?

PREVIOUS NEXT

LOG IN

Figure 18: Pagina per visualizzare la classifica globale

Sign up into Checkers

Username	<input type="text" value="Create your username"/>
Name	<input type="text" value="Insert name"/>
Last name	<input type="text" value="Insert last name"/>
Insert your mail	<input type="text" value="Insert your mail"/>
Password	<input type="text" value="Insert password"/>
Confirm Password	<input type="text" value="Confirm password"/>
<input type="button" value="SIGN UP"/>	

LOG IN

Figure 19: Pagina per la registrazione di un utente

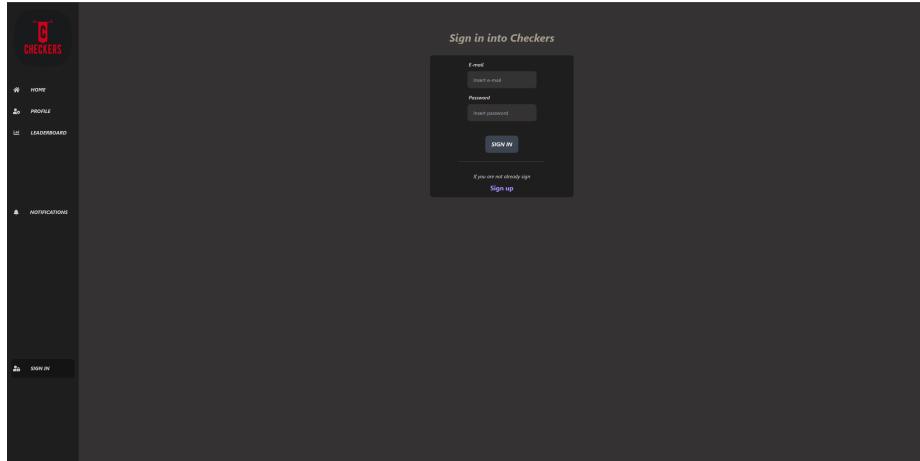


Figure 20: Pagina per accedere al sito

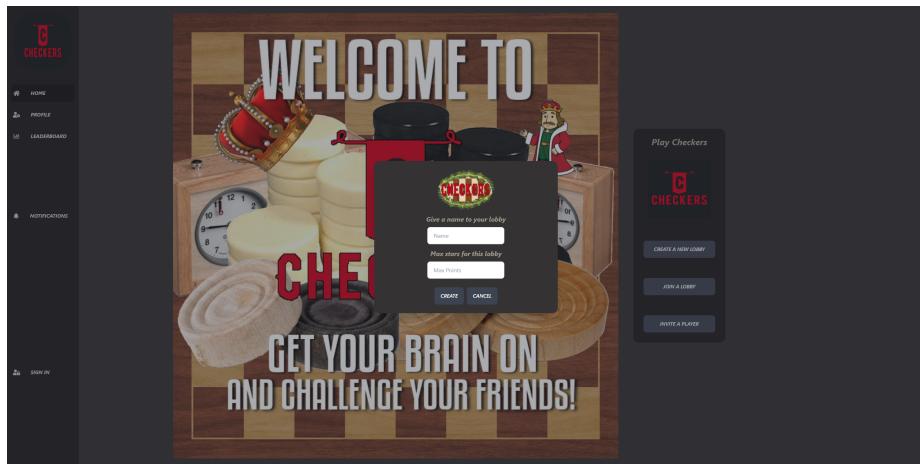


Figure 21: Modale per la creazione di una stanza

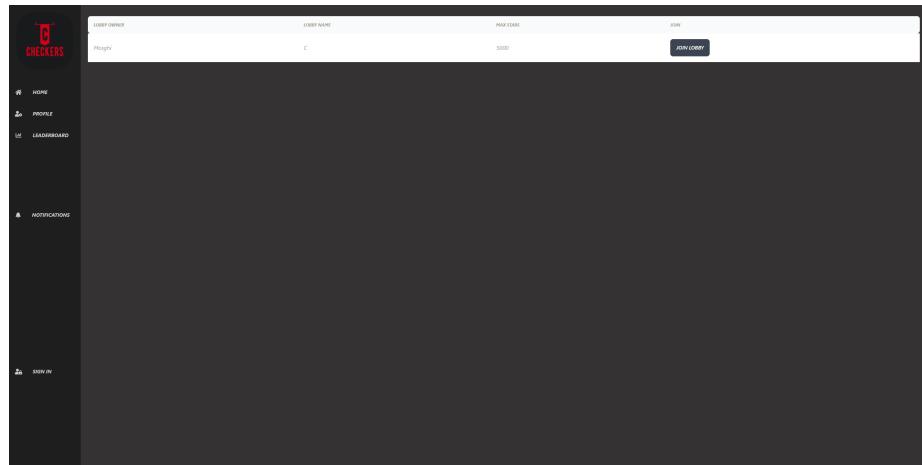


Figure 22: Pagina per la visualizzazione di tutte le lobby a cui è possibile connettersi

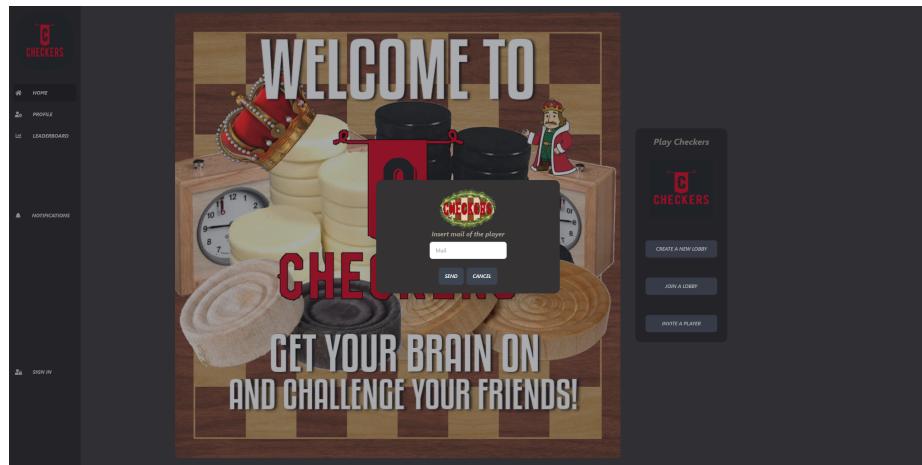


Figure 23: Modale per invitare un altro giocatore

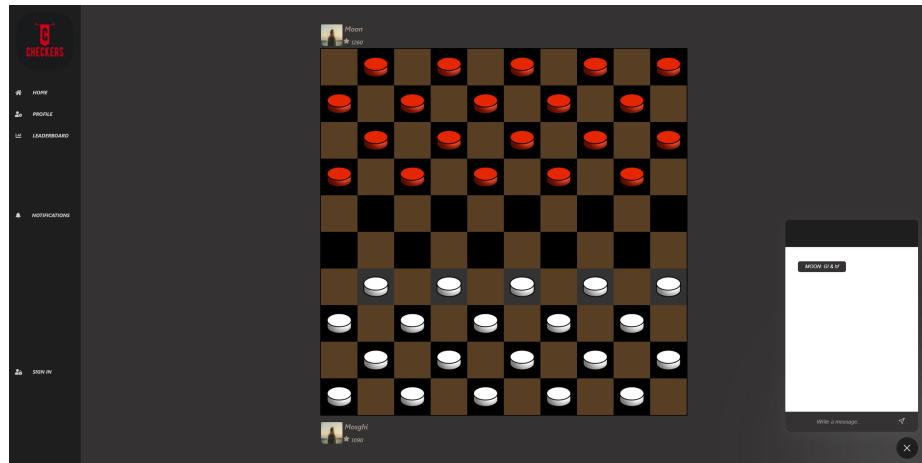


Figure 24: Pagina di gioco

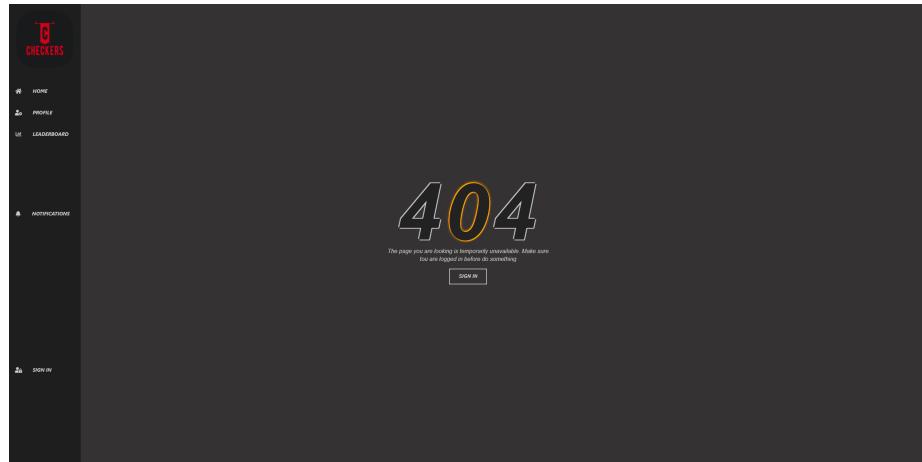


Figure 25: Pagina quando si riceve un errore 404

Bibliography

- [1] Anthony Strittmatter, Uwe Sunde, and Dainis Zegners. Life cycle patterns of cognitive performance over the long run. *Proceedings of the National Academy of Sciences*, 117(44):27255–27261, oct 2020.